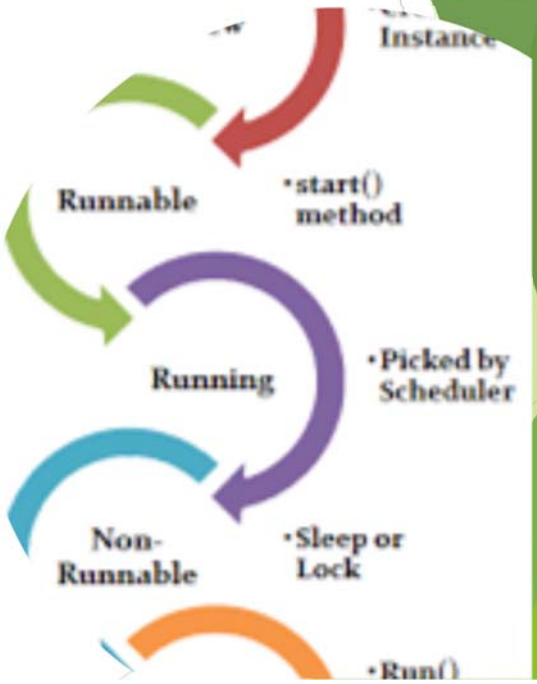
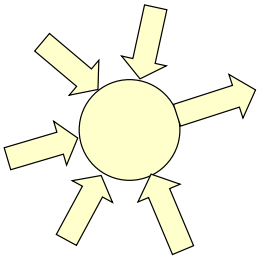
Thread, les processus légers



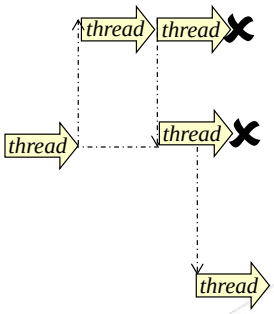
Synchronisation

- Gestion des accès concurrents à des données communes de deux *threads*
  - ⇨ risque de modification non cohérente.
- Pour résoudre ce problème, on peut bloquer l'accès à certaines données par un système de verrouillage.

Synchronisation autour d'une ressource

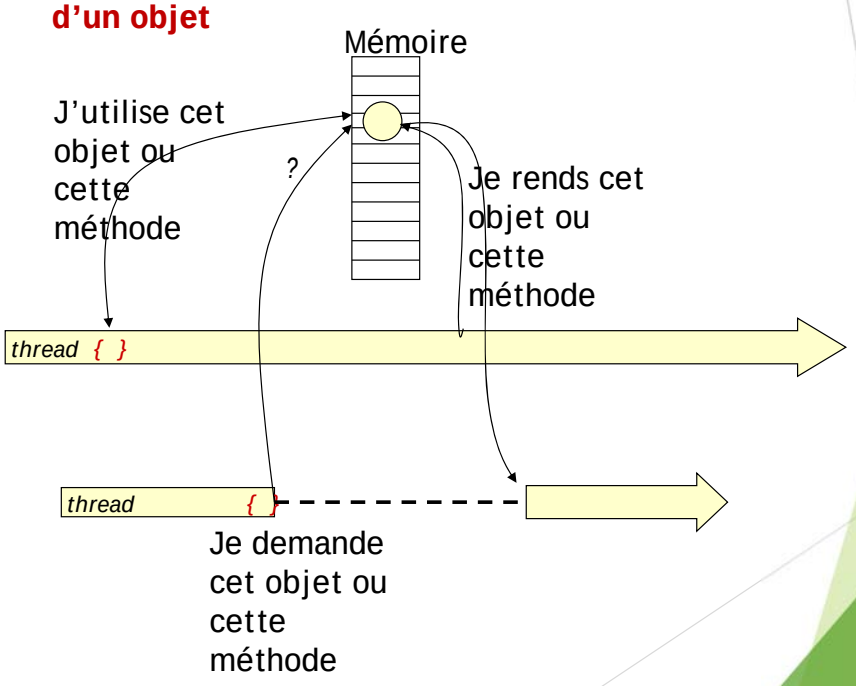


Synchronisation temporelle



synchronisation

Synchronisation autour d'un objet



synchronisation autour d'un objet

- Poser un verrou sur un objet pendant l'exécution d'un certain nombre d'instructions.

```
public class Synchronisation extends Thread
{
    static Integer i;
    int max;
    String nom;
    public Synchronisation(String nom, int max)
    {
        i = new Integer(0);
        this.max = max;
        this.nom = nom;
    }
    public void run()
    {
        synchronized(i)
        {
            while (i.intValue() < max)
            {
                try { sleep(100);}
                catch (Exception e) { System.err.println(e);}
                System.out.println(nom + " modifie i.");
                i = new Integer(i.intValue() + 1);
                System.out.println(i.intValue());
            }
            System.out.println(nom + " est fini !");
        }
    }
    public static void main(String[] args)
    {
        new Synchronisation("t1", 50).start();
        new Synchronisation("t2", 100).start();    }}
```

Choisir un objet partagé sinon : inutile

La portée de ce verrou est définie au sein du *thread* par un bloc identifié par le mot-clé ***synchronized*** associé à une référence vers l'objet à verrouiller.

Les autres *threads* ne pourront pas effectuer d'action sur cet objet et attendront la levée du verrou avant de poursuivre leurs traitements.

synchronisation par les méthodes

- **Verrouillage de l'accès à une méthode ;**
- **Deux objets qui en disposent ne peuvent y recourir simultanément.**

```
public class MonImplDeRunnable implements Runnable
{
    public void run()
    {
        for (int i = 0; i < 5; i++)  move();
    }
    synchronized private void move()
    {
        String nom = Thread.currentThread().getName();
        System.out.println(nom + " entre.");
        try {Thread.sleep(1000);
          } catch (InterruptedException e) {e.printStackTrace();}
        System.out.println(nom + " sort.");
    }
    public static void main(String[] args)
    {
        MonImplDeRunnable o = new MonImplDeRunnable();
        new Thread(o).start();
        new Thread(o).start();
    }
}
```

```
Thread-0 entre.
Thread-0 sort.
Thread-0 entre.
Thread-0 sort.
Thread-0 entre.
Thread-0 sort.
Thread-1 entre.
Thread-1 sort.
Thread-0 entre.
Thread-0 sort.
Thread-1 entre.
Thread-1 sort.
Thread-1 entre.
Thread-1 sort.
Thread-0 entre.
Thread-0 sort.
Thread-1 entre.
Thread-1 sort.
Thread-1 entre.
Thread-1 sort.
```

ces deux *thread* ne pourront exécuter simultanément la méthode *synchronized move()*

- Dans le cas d'une méthode statique, tous les *threads* issus de la classe sont affectés et ne peuvent exécuter simultanément la méthode.

synchronisation temporelle : wait et notify

- *wait()* / *notify()* : *Rendez-vous* entre Thread : en attente d'un signal sur un objet partagé

- Attente entre Threads : *o.wait()*.
  - Suspend de l'exécution du *thread* courant.
  - Le *thread* attend que l'objet *o* émette une *notification* afin de reprendre son exécution.
  - *wait(long duree)*. Ajoute une durée maximale d'attente à l'issue de laquelle, notification ou pas, le *thread* redémarre.

- Notifier un *thread* : *notify()*
  - Cette méthode appelée sur un objet *o* réactive un *thread* en attente suite à l'appel de la méthode *wait()* sur cet objet *o*.
  - S'il y a plusieurs *threads* à réactiver (*i.e.* en attente en même temps sur le même objet *o*), le choix du *thread* à réveiller est arbitraire et dépend de l'implémentation de la JVM.
  - *notifyAll()*: Cette méthode appelée sur un objet *o* réactive tous les *threads* ayant effectué un appel de la méthode *wait()* sur ce même objet *o*.

wait() et notify()

```
public class MonImplDeRunnable implements Runnable
{
    public void run()
    {
        Thread t = Thread.currentThread();
        //t: thread en cours d'exécution.
        try
        {
    synchronized(t)
            {
                t.wait();
// attente infinie réveil par un notify
            }
            System.out.println(t.getName() + " a fini d'attendre.");
        }
        catch (Exception e) {e.printStackTrace(System.err);}
        finally { System.out.println("Fin de " + t.getName()); }
    }

 public static void main(String[] args)
    {   try {
            Thread t = new Thread(new MonImplDeRunnable());
            t.start();
            System.out.println("On attend 3 secondes...");
            Thread.sleep(3000);
            System.out.println("On réveille " + t.getName());
            synchronized(t)
              { t.notify(); }        }
```

Attention : ces méthodes ne peuvent être appelées que sur un objet verrouillé (bloc ***synchronized)***
Sinon : ***IllegalMonitorStateException***

wait() et notify()

- La méthode *wait()* libère temporairement le verrou sur l'objet synchronisé. Une fois que le *thread* est réactivé (à l'aide de *notify()*, de *notifyAll()* ou à la fin de la durée maximale d'attente), le verrou est acquis de nouveau.
- À noter également que les méthodes *wait()*, *notify()* et *notifyAll()* ne sont pas mises en attente lorsqu'elles sont appelées par un *thread* sur un objet verrouillé par un autre *thread*.
- Suite à l'appel de la méthode *wait()*, un *thread* est dans un état *waiting* (état d'attente).

wait() et notify()

```
public class Starter extends Thread
{
     Integer dep;


Public Starter(Interger dep)
         {
         this.dep=dep;
         }

    public void run()
    {
      for (int i = 5; i > 0; i--)  syso("bla bla " + i);
        try
        {
         synchronized(dep)
              {
               dep.notifyAll();
              }
}
         catch (Exception e) {e.printStackTrace(System.err);}
         finally { System.out.println("Fin de " + t.getName()); }
    }
```


wait() et notify()

```
public class Sprinter  extends Thread
{
    Integer dep;


Public Sprinter(Integer dep)
         {
         this.dep=dep;
         }

   public void run()
   {
    try
        {
         synchronized(dep)
              {
               dep.wait();
              }
}
         catch (Exception e) {e.printStackTrace(System.err);}
         finally { System.out.println("Fin de " + t.getName()); }
   }
```

wait() et notify()

```
public class Sprinter  extends Thread
{
     Integer dep;


Public Sprinter(Integer dep)
          {
          this.dep=dep;
          }

    public void run()
    {
     try
         {
          synchronized(dep)
                {
                 dep.wait();
                }
}
          catch (Exception e) {e.printStackTrace(System.err);}
          finally { System.out.println("Fin de " + t.getName()); }
    }
```